Metody Kompilacji
Wykład 1
Wstęp

Literatura:

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman:
Compilers: Princiles, Techniques, and Tools. Addison-Wesley 1986, ISBN 0-201-10088-6

Literatura:

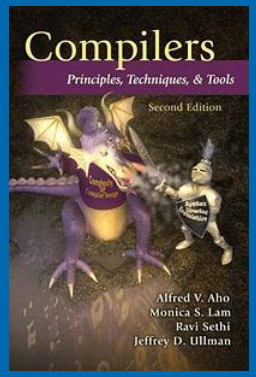
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman:
Compilers: Princiles, Techniques, and Tools. Addison-Wesley 2007, ISBN 0-321-48681-1

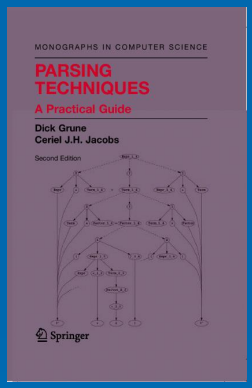
Literatura:

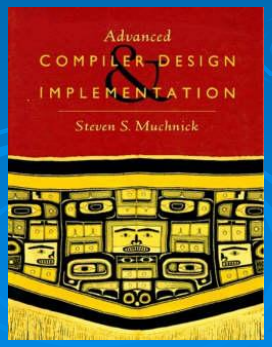
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Kompilatory : reguły, metody i narzędzia*. Warszawa: WNT, 2002. ISBN 83-204-2656-1.

Literatura:

- **Compilers: Principles, Techniques, and Tools**, Aho, Sethi and Ullman
  - http://dragonbook.stanford.edu/
- **Parsing Techniques**, Grune and Jacobs
- **Advanced Compiler Design and Implementation**, Muchnik

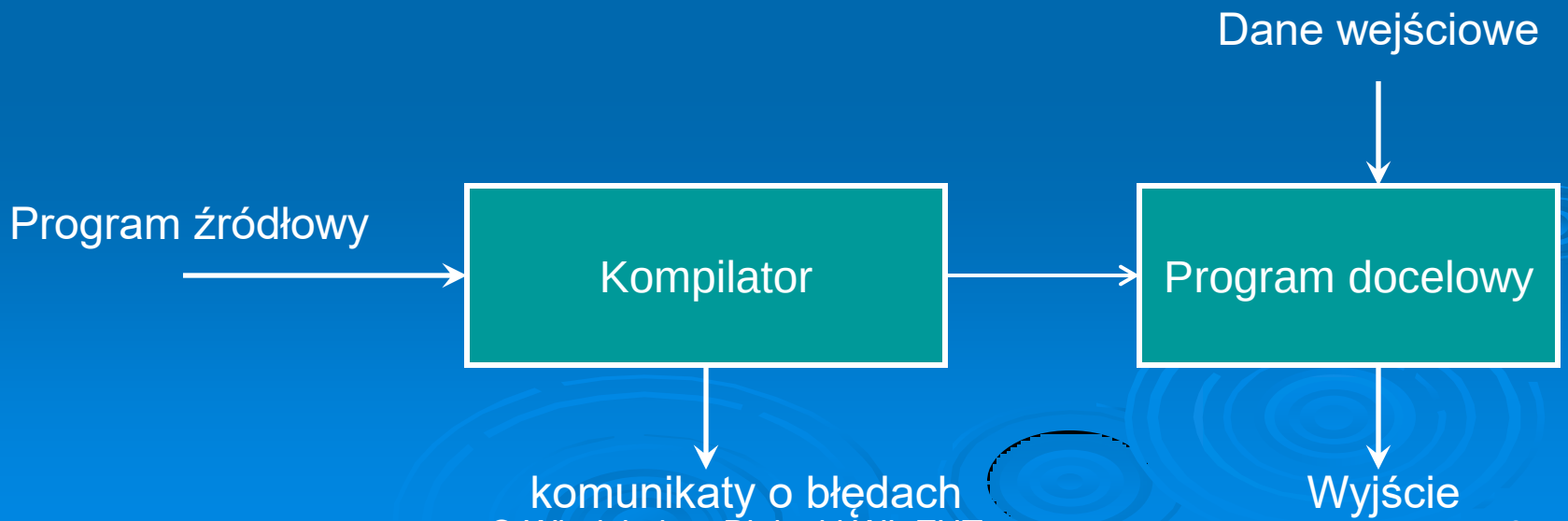




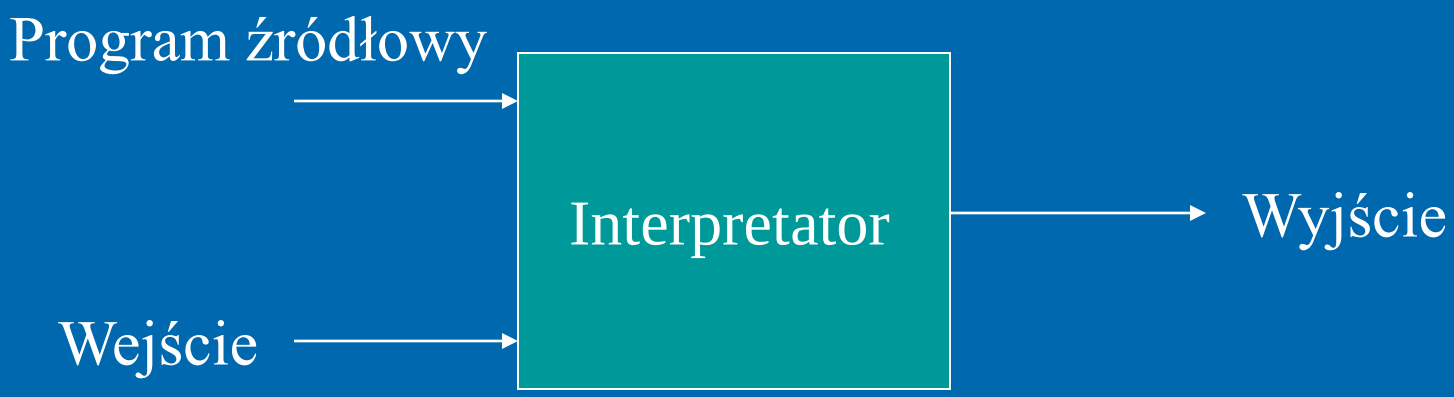


Kompilator

- Kompilator jest to program, który odczytuje program w języku źródłowym i tłumaczy go na program równoważny w języku docelowym.



Interpretator



Interpretator, zamiast produkowania programu docelowego, bezpośrednio wykonuje czynności określone w programie źródłowym.

Kompilatory

- Program docelowy, produkowany przez kompilator, jest zwykle znacznie szybszy niż proces produkowania wyniku przez interpretator.
- Interpretator, jednak zazwyczaj daje lepszą diagnostykę błędów niż kompilator, ponieważ wykonuje instrukcję programu źródłowego instrukcja po instrukcji.

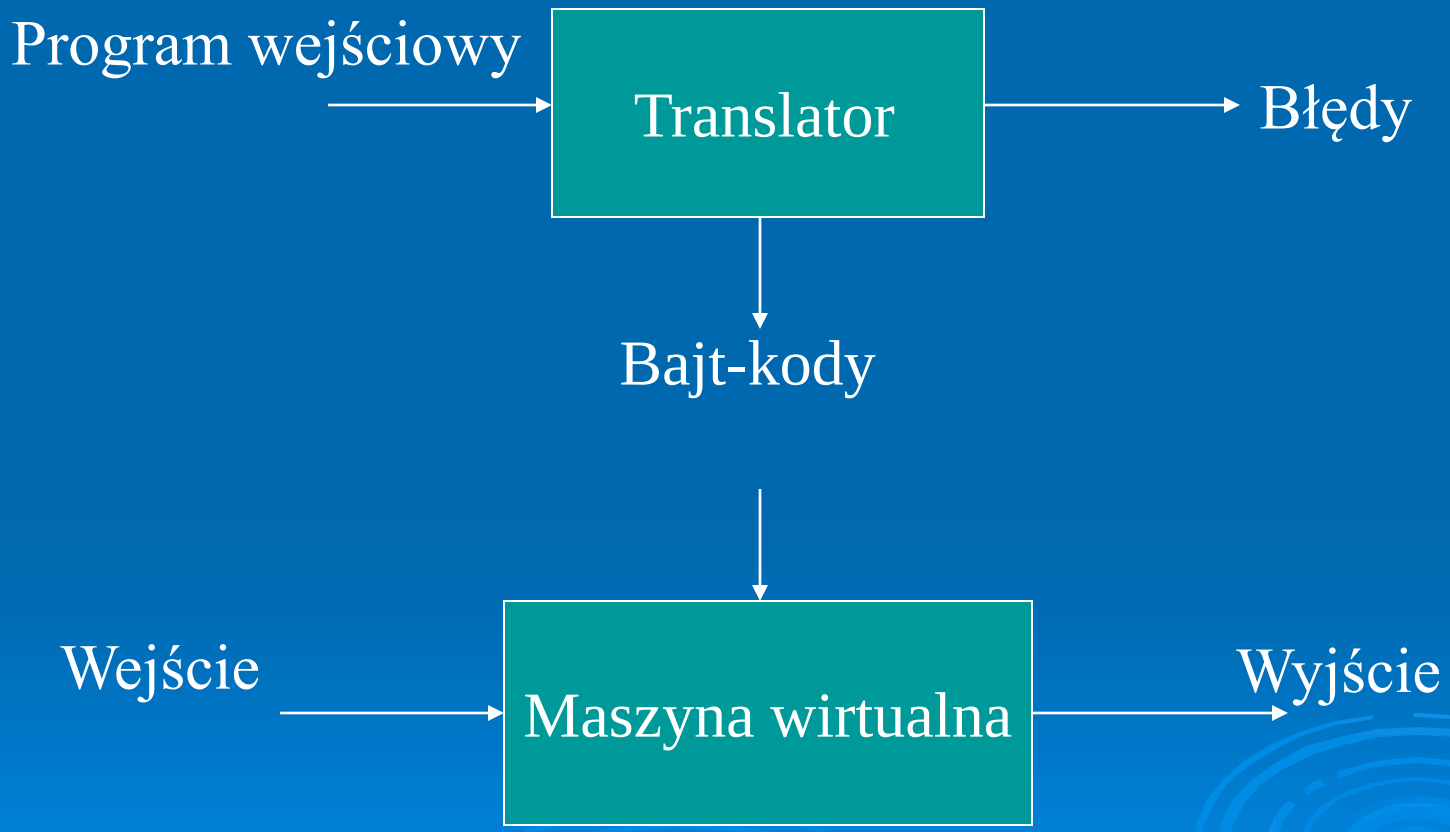
Kompilatory

- Ważną rolą kompilatora jest zgłaszanie wszelkich błędów w programie źródłowym, które są wykrywane podczas procesu tłumaczenia.

Kompilatory

- Wirtualne procesory języka Java łączą kompilację i interpretację.
- Program źródłowy w języku Java jest najpierw kompilowany do postaci pośredniej zwanej bajt-kodami (*bytecodes)*.
- Bajt-kody następnie są interpretowane przez maszynę wirtualną.

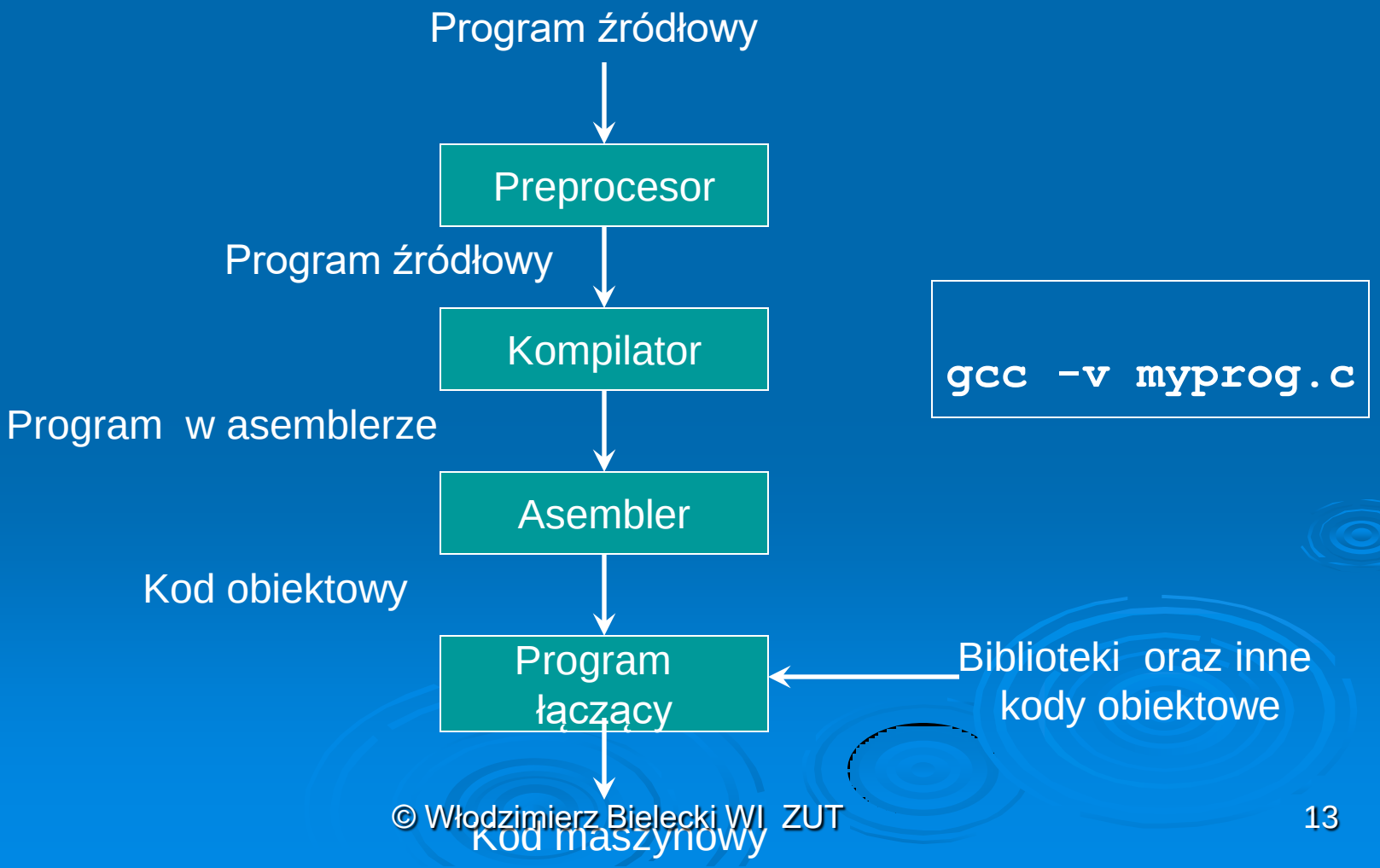
Kompilator hybrydowy



Kompilatory

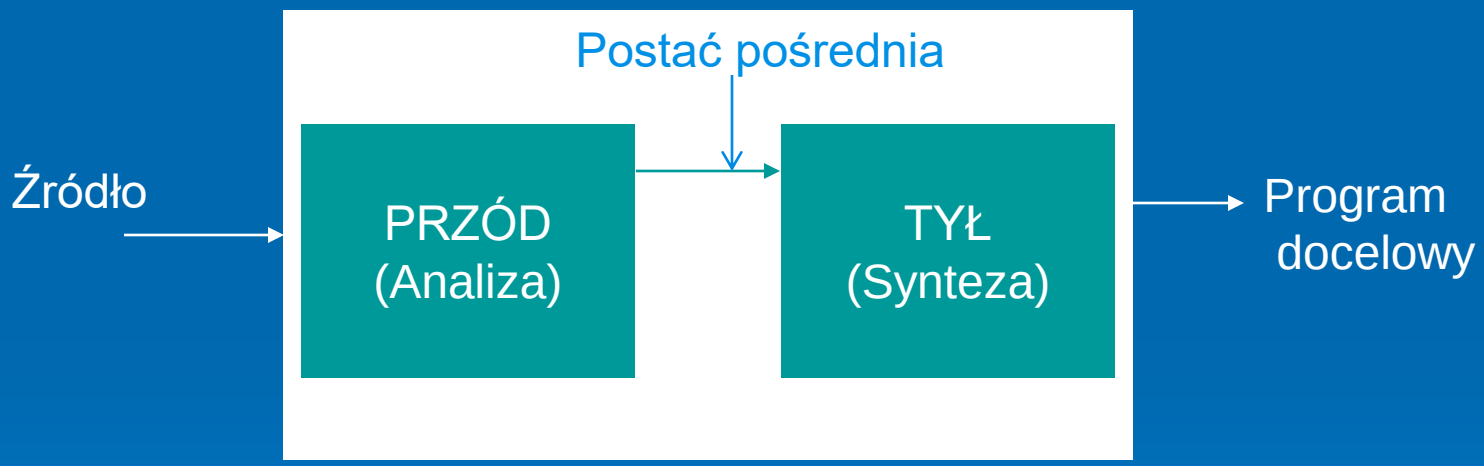
- Oprócz kompilatora, kilka innych programów może być wymagane, aby utworzyć wykonywalny program docelowy.

Kompilatory




```
gcc -v myprog.c
```

Architektura kompilatora



Architektura kompilatora

Analiza leksykalna

- Analizator leksykalny czyta znaki z programu źródłowego i grupuje je w sekwencje, które reprezentują leksemy.
- Dla każdego leksemu, analizator leksykalny produkuje token o postaci:

<token-name, attribute-value>

Analiza leksykalna

- Na przykład, załóżmy, że program źródłowy zawiera instrukcję przypisania:

  position := i n i t i a l + r a t e * 60

- Analizator leksykalny produkuje następujący wynik:

Analiza leksykalna

- 1. position jest leksemem, dla którego jest tworzony token:

  <id, 1>

  gdzie id jest to symbol abstrakcyjny oznaczający identyfikator; 1 jest adresem, pod którym tablica symboli przechowuje leksem position oraz dodatkowe atrybuty, na przykład typ danych.

Analiza leksykalna

- 2. Symbol przypisania := jest leksemem, dla którego jest produkowany token <:= >.

- Ponieważ ten token nie wymaga atrybutu, drugi składnik jest pominięty.

Analiza leksykalna

- 3. initial jest leksemem, dla którego jest tworzony token <id, 2>; 2 jest adresem, pod którym tablica symboli przechowuje leksem initial.

Analiza leksykalna

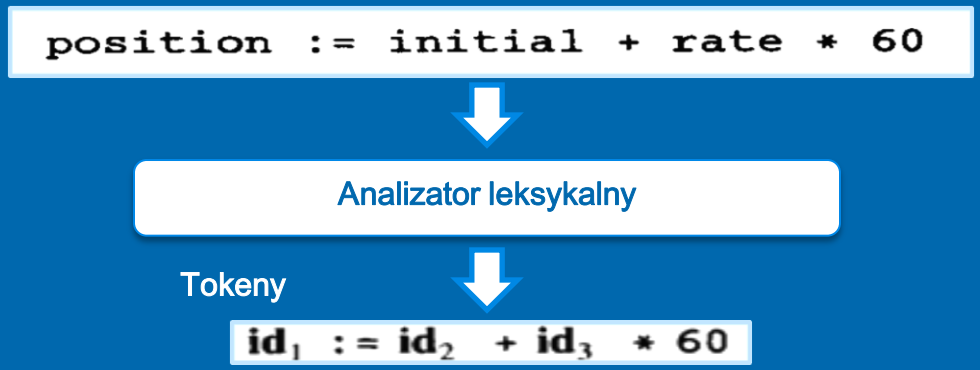
- 4. + jest leksemem, dla którego jest produkowany token < + >.
- 5. rate jest leksemem, dla którego jest tworzony token <id, 3>, 3 jest adresem, pod którym tablica symboli przechowuje leksem rate.

Analiza leksykalna

- 6. * jest leksemem odwzorowywanym na token < * > .
- 7. 60 jest leksemem odwzorowywanym na token < 60 >.

60 jest liczbą całkowitą

Analiza leksykalna



Wyżej jest wynik produkowany przez analizator leksykalny dla instrukcji:

position:= i n i t i a l + r a t e * 60

Analiza syntaktyczna

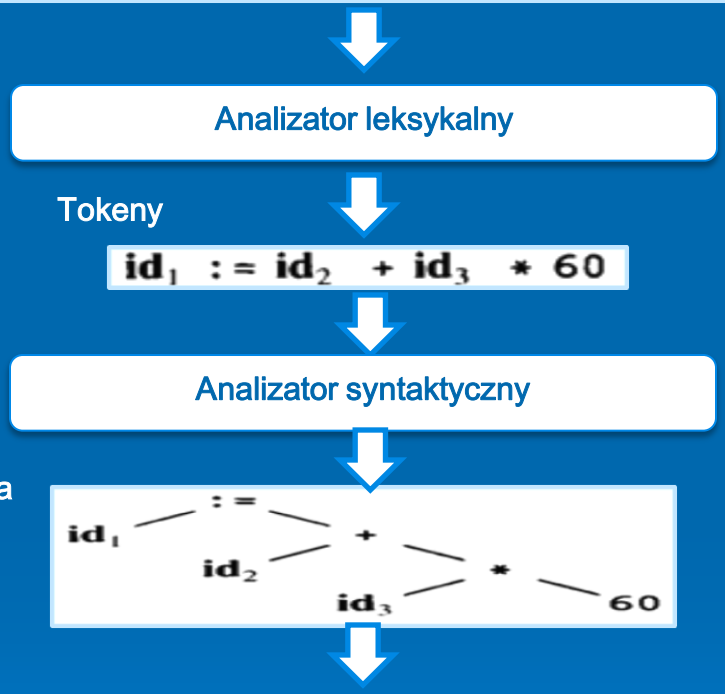
- Parser wykorzystuje pierwsze składniki tokenów, produkowane przez analizator leksykalny, aby utworzyć reprezentację pośrednią, która przedstawia strukturę gramatyczną strumienia tokenów.

Analiza syntaktyczna

- Typową reprezentacją składni jest drzewo syntaktyczne, w którym każdy węzeł wewnętrzny reprezentuje operację, natomiast dzieci węzła stanowią argumenty operacji.
- W oparciu o utworzone drzewo, parser decyduje czy składnia programu jest poprawna czy nie.
- Na wyjściu parsera mamy wynik pokazany na następnym slajdzie.

Analiza syntaktyczna

```
position := initial + rate * 60
```



Analiza semantyczna

- Analizator semantyczny korzysta z drzewa parsowania, wykorzystuje informacje przechowywane w tablicy symboli i sprawdza program źródłowy pod względem spójności semantycznej, zdefiniowanej przez język programowania.

Analiza semantyczna

- Ponadto gromadzi informacje o typach zmiennych i zapisuje je w drzewie parsowania lub tablicy symboli do wykorzystania podczas kolejnych etapów generacji kodu pośredniego.

Analiza semantyczna

- Ważną częścią analizy semantycznej jest kontrola typów, kompilator sprawdza czy każdy operator ma dopasowane argumenty.

- Na przykład, wiele języków programowania wymagają, żeby indeksy tablicy były liczbami całkowitymi; kompilator musi zgłosić błąd, jeśli liczba zmiennoprzecinkowa jest używana do reprezentacji indeksu tablicy.

Analiza semantyczna

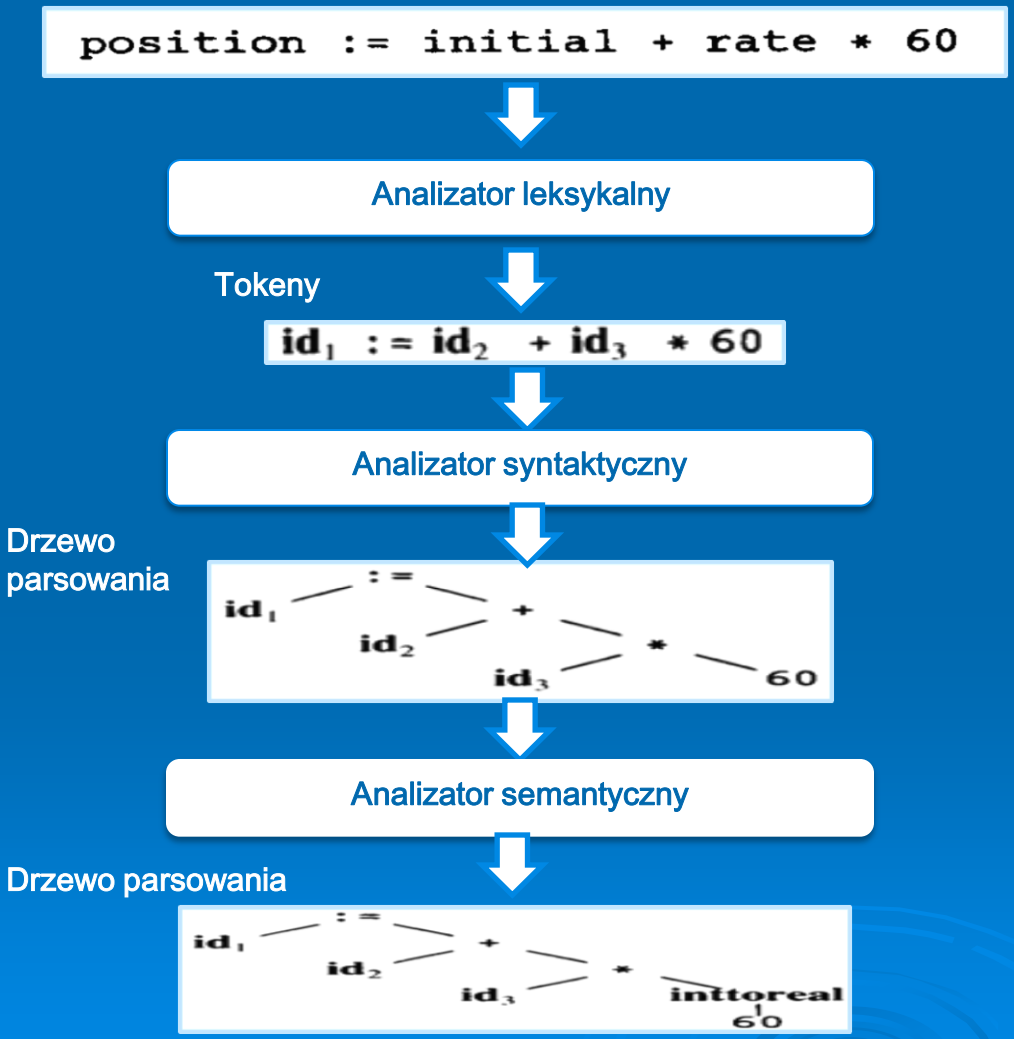
- Specyfikacja języka może pozwolić na konwersję typów, znaną jako wymuszenie (*coercion*).
- Na przykład, operator arytmetyczny może być zastosowany do pary liczb całkowitych lub pary liczb zmiennoprzecinkowych.

Analiza semantyczna

- Jeśli operandy nie należą do tego samego typu danych, to jeden z nich może być konwertowany do typu drugiego operandu.

- Na rysunku na następnym slajdzie, operator *inttoreal* konwertuje liczbę całkowitą na liczbę zmiennoprzecinkową.

Analiza semantyczna



Generacja kodu pośredniego

- W procesie tłumaczenia programu źródłowego na kod docelowy, kompilator może utworzyć jedną lub kilka reprezentacji pośrednich, które mogą mieć różne formy.

- Na przykład, drzewa składniowe są popularną formą reprezentacji pośredniej.

Generacja kodu pośredniego

- Drugą popularną formą jest kod trójadresowy:

t1 = inttoreal(60)

t 2 = id3 * t1

t 3 = id2 + t 2

id1 = t 3

Generacja kodu pośredniego

```
position := initial + rate * 60
```

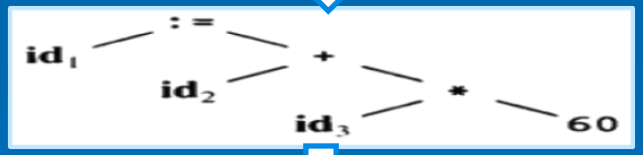
Analizator leksykalny

Tokeny

$id_1 := id_2 + id_3 * 60$

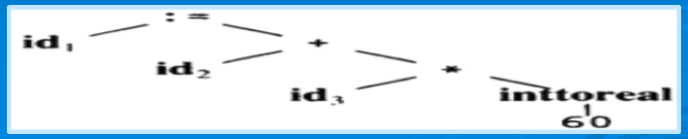
Analizator syntaktyczny

Drzewo parsowania



Analizator semantyczny

Drzewo parsowania



Generator kodu pośredniego

Kod pośredni

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Optymalizacja kodu

- Optymalizacja kodu polega na redukcji liczby instrukcji i/lub zmniejszeniu zapotrzebowania na pamięć (zmniejszenie liczby zmiennych tymczasowych).
- Optymalizacja kodu może być fazą opcjonalną.

Optymalizacja kodu

```
position := initial + rate * 60
```

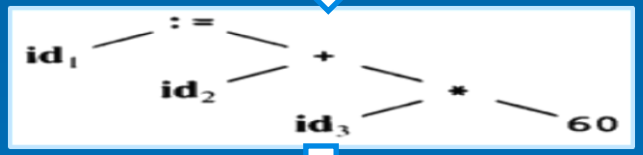
Analizator leksykalny

Tokeny

$id_1 := id_2 + id_3 * 60$

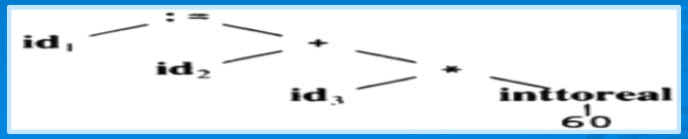
Analizator syntaktyczny

Drzewo parsowania



Analizator semantyczny

Drzewo parsowania



Generator kodu pośredniego

Kod pośredni

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Optymalizator kodu

Kod zoptymalizowany

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Generacja kodu

- Generator kodu tłumaczy reprezentację pośrednią programu na program w języku docelowym.
- Jeśli programem docelowym ma być kod maszynowy, to generator kodu musi przydzielić pamięć(rejestry, pamięć operacyjną) dla każdej zmiennej zdeklarowanej w programie źródłowym.

Generacja kodu

- Następnie każda instrukcja postaci pośredniej jest tłumaczona na sekwencję instrukcji maszynowych.
- Aspektem kluczowym generowania kodu maszynowego jest optymalny przydział rejestrów do przechowywania zmiennych.

Generacja kodu

```
position := initial + rate * 60
```

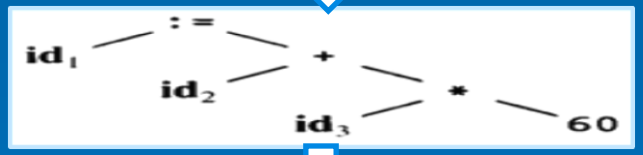
Analizator leksykalny

Tokeny

$id_1 := id_2 + id_3 * 60$

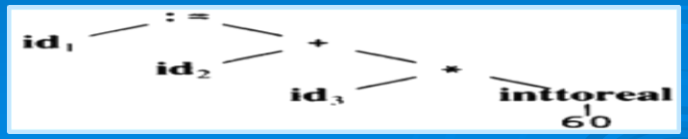
Analizator syntaktyczny

Drzewo parsowania



Analizator semantyczny

Drzewo parsowania



Generator kodu pośredniego

Kod pośredni

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Optymalizator kodu

Kod zoptymalizowany

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Generator kodu docelowego

Kod docelowy

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

Tablice symboli

- Tablica symboli jest strukturą danych zawierająca rekord dla każdej nazwy zmiennej wraz z polami do przechowywania atrybutów tej zmiennej.
- Tablica symboli powinna być zaprojektowana tak, aby kompilator mógł szybko znaleźć rekordy dla każdej nazwy oraz szybko zapisać lub pobrać dane z tego rekordu.

Grupowanie faz kompilacji

- Przedstawione fazy kompilatora pokazują logiczną organizację kompilatora.

- W implementacji kompilatora, fazy te mogą być grupowane w jedną większą fazę.

Grupowanie faz kompilacji

- Na przykład, analiza leksykalna, analiza syntaktyczna i analiza semantyczna mogą być połączone w jedną fazę: w tej fazie czynności wszystkich tych analiz są wykonywane jednocześnie pod kontrolą analizatora syntaktycznego (kompilacja sterowana składnią).

Dziękuję za uwagę